Characterizing Compatibility of Timed Choreography.

Nawal Guermouche, Claude Godart

Characterizing Compatibility of Timed Choreography

Nawal Guermouche[1, 2, 3] and Claude Godart[3]

[1] CNRS ; LAAS ; 7 avenue colonel Roche, F-31077 Toulouse, France
[2] Université de Toulouse ; UPS, INSA, INP, ISAE, UT1, UTM, LAAS ; F-31077 Toulouse, France
[3] LORIA-Nancy University, UMR 7503
BP 239, Vandoeuvre-les-Nancy, France

ABSTRACT:

Web services are the main pillar of the Service Oriented Computing (SOC) paradigm which enables application integration within and across business organizations. One of the most important features of Web services is the idea of *choreography* which allows to capture collaborative processes involving multiple services. In this context, *compatibility analysis of choreography* is a central point to investigate. We mean by compatibility of a choreography the capability of a set of Web services of actually interacting by exchanging messages in a safe way. Whether a set of services are compatible depends not only on their sequences of messages but also on some *quantitative properties* such as *timed properties*. In this paper, we investigate a model checking based approach that deals with checking the *compatibility of a choreography* in which Web services support *asynchronous timed communications*.

KEY WORDS:
Asynchronous Web services, Choreography analysis, compatibility analysis, timed properties

INTRODUCTION

The evolution of computer science technologies has given life to many paradigms such as the Service Oriented Computing (SOC) paradigm (Alonso and al. 2004, Benatallah and al. 2007, Dijkman and al. 2004). In this latter, Web services are the main pillar. Based on standard interfaces, Web services facilitate application-to-application interactions. This advantageous property of Web services gives rise to several important concepts such as the notion of choreography. Such a feature offers the possibility to capture collaborative processes involving multiple services where the interactions between these services are seen from a global perspective. In this context, one of the important elements is the compatibility analysis. By compatibility we mean the capability of a set of services of actually fulfilling successful interactions by exchanging messages.

In the last few years, some works have investigated the compatibility problem of two Web services: a client and a provider service (Bordeaux and al. 2004, Benatallah and al. 2005-a, Benatallah and al. 2005-b, Ponge and al. 2007, Guermouche and al. 2008-a). In all these works, the authors deal with services that support *synchronous communications*. In that case, to characterize the compatibility class of two services, the authors check if each input (resp. output) message of a service corresponds to an output (resp. input) message of the other service in the same order (i.e., the services are synchronized over messages). However, the nature of distributed systems and particularly of Web services can be asynchronous, hence the problem of the applicability of these approaches which are very restrictive in real application scenarios is still open. To overcome such a limitation, in this paper we tackle the problem of analyzing the

compatibility of a choreography in which Web services support *asynchronous* communications. In an *asynchronous* communication, when a message is sent, it is inserted into a bounded message queue, and the receiver consumes (i.e. receives) the message while it is available in the queue.

On the other side, it is commonly agreed that in general the interaction of Web services and in particular the compatibility of Web services depends not only on the supported sequences of messages but there are other crucial *quantitative properties* such as *timed properties* (Benatallah and al. 2005-a, Benatallah and al. 2005-b, Kazhamiakin and al. 2006-a, Kazhamiakin and al. 2006-b, Ponge and al. 2007, Guermouche and al. 2008-a). We mean by *timed properties* the required delays to exchange messages (e.g., in an e-government application, a prefecture can send its final decision to grant an handicapped pension to a requester after 7 days and within 14 days). When services are interacting, their timed properties can be conflicting. The existing works cannot discover all the eventual timed conflicts since the authors rely on the principle of synchronizing the services over messages (Bordeaux and al. 2004, Benatallah and al. 2005-a, Benatallah and al. 2005-b, Ponge and al. 2007, Guermouche and al. 2008-a).

In this paper, we propose a framework for analyzing a choreography compatibility in the context of asynchronous communicating services. In this framework we take into account data flow that can be involved when exchanging messages. Furthermore, we consider constraints over data and timed properties that specify delays concerning message exchanges. By studying the possible impacts of timed properties on a choreography, we remarked that when Web services are interacting together, *implicit timed dependencies* can be derived from different timed properties of the different services. Such dependencies can give rise to *implicit timed conflicts*. To discover deadlocks due to timed conflicts, we first study the possibility to apply the existing compatibility approaches of synchronous services (Benatallah and al. 2005-a, Ponge and al. 2007, Guermouche and al. 2008-a, Guermouche and al. 2008-d), but we concluded that the existing approaches are inadequate to discover all the eventual timed deadlocks since the authors rely on synchronizing the services over messages. In order to catch all the possible timed deadlocks, we propose a set of model checking based primitives.

One of the important ingredients we need in a compatibility framework is the Web services description behavior. The behavior of a Web service specifies the sequences of messages the service supports, the involved data types, and the associated timed requirements. The timed behavior of a Web service specifies the *timed conversational protocol* (for short we say conversational protocol). For compatibility analysis, we have chosen to model a conversational protocol as a finite state machine (FSM) specification. This kind of formal representation has been already used in a series of work (Bultan and al. 2003, Benatallan and al. 2005-a, Berardi and al. 2005, Ponge and al. 2007, Anca and al. 2007, Guermouche and al. 2008-a) and seems adequate. In fact, a state machine based model is suitable to describe reactive behaviors (Benatallah and al. 2005-b), it is fairly easy to understand, and at the same time it is expressive enough to model the properties we consider. In addition, we rely on clocks as defined in standard timed automata (Alur and al. 1994).

To summarize, in this paper we make the following contributions: (1) We propose an asynchronous model of Web services that gathers messages, data types, data constraints, and timed requirements. (2) We propose an abstraction process that allows to apply a model checking to analyze asynchronous Web services. (3) Unlike the existing compatibility frameworks, we propose primitives for analyzing and characterizing the compatibility class of a choreography in which the Web services support asynchronous timed communications.

The reminder of the paper is organized as follows. Next section presents an e-government case study that we use to show the related issues of the proposed approach. Then, we present how we model the timed behavior of Web services. For better understanding, we discuss informally and intuitively the timed compatibility problem of a choreography. In order to be able to handle asynchronous services by UPPAAL, we present a set of abstractions and transformations. After that, we present our formal choreography compatibility investigations. Then, we discuss related work. Before concluding, we describe experimentation step.

CASE STUDY: E-GOVERNMENT APPLICATION

Let us present a part of an e-government application inspired from (Mecella and al. 2001) to illustrate our approach. The goal of the e-government application we consider is to manage handicapped pension requests. Such a request involves three Web services: (1) *prefecture service* (PS) (2) *health authority* service (HAS), and (3) *town hall* service (TH).

The high level choreography model of the process is depicted in Figure 1. A citizen can apply for pension. To do so, the citizen asks the corresponding form from the prefecture. Once the form is received, the citizen must return the form filled. The prefecture solicits the medical entity to examine the requester. The health authority negotiates a date of an appointment to examine the citizen. After the examination, the health authority service sends a medical report to the prefecture. On the other side, the prefecture asks the town hall to deliver the domiciliation attestation. After studying the received file, the medical report and the domiciliation attestation, the prefecture sends the notification of the final decision to the citizen.

The interaction between these partners is constrained by timed requirements.

- Once the health authority service proposes meeting dates to the citizen, this latter must send the filled form within 24 hours.
- The prefecture requires at least 48 hours and at most 96 hours from receiving the file from the requester to notify the citizen by the final decision.
- The medical report can be sent to the prefecture after at least 120 hours and at most 168 hours from receiving the medical verification request.

Besides, Web services can be constrained by requirements on data. For example, the prefecture that specifies the pension application can be considered if the applicant is at least 17 years old.

The Web services we consider can support asynchronous communications. The first issue we deal with is how to analyze the compatibility of a choreography in which the Web services are asynchronous? Moreover, the behavior of the Web services might be constrained by constraints over data and timed requirements.

As in a choreography with several services possibly asynchronous, the timed properties are local and are mutually independent, hence, when the services are interacting together, timed deadlocks can arise. To assert the global interaction between the Web services (i.e., ensure that the choreography is deadlock free), we need primitives that consider timed properties when analyzing the compatibility of Web services. The second issue we need to handle is how to consider data constraints and timed properties together when analyzing the compatibility of a choreography?

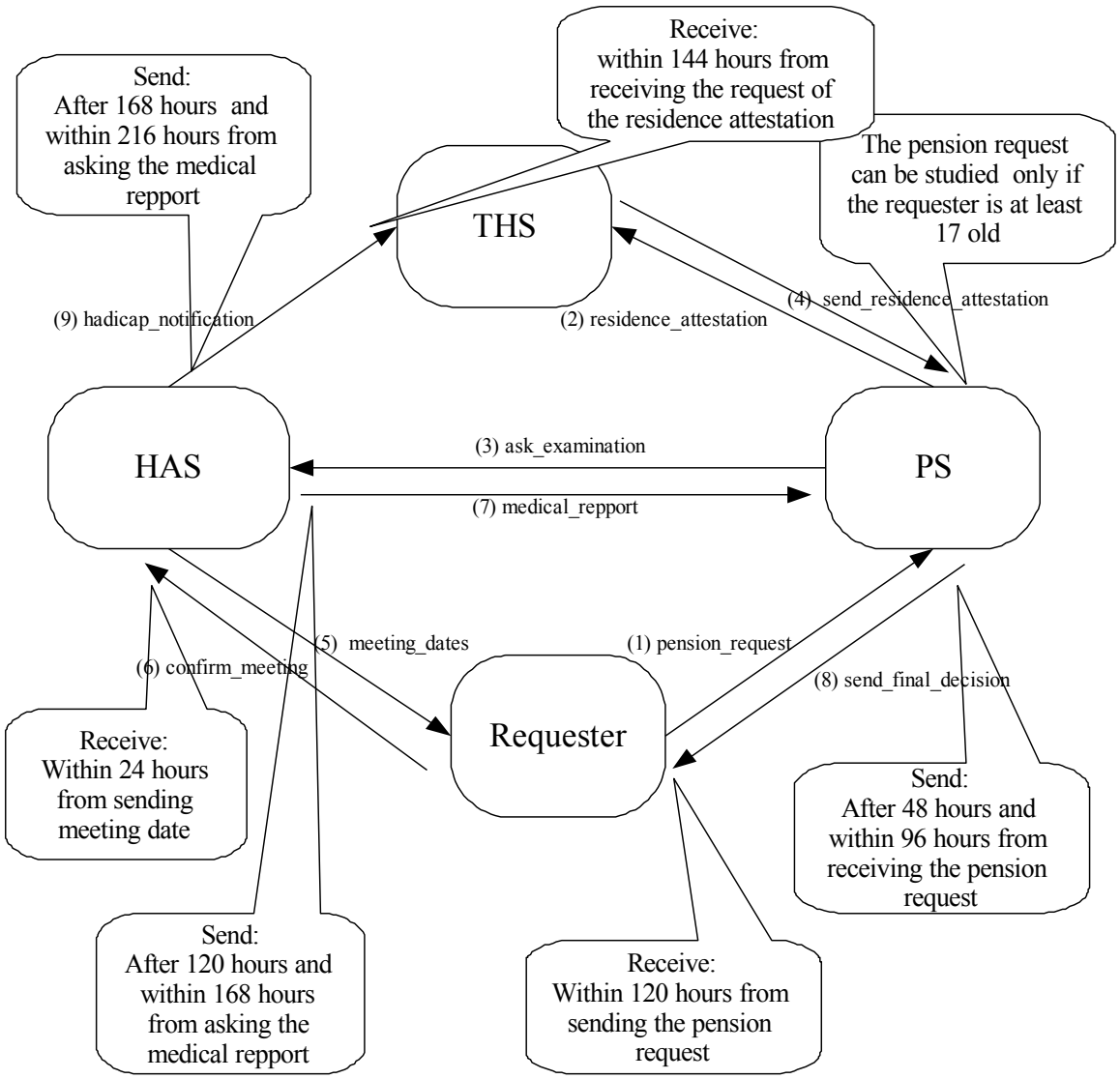

Figure 1. Global view of the e-government application

MODELING TIMED BEHAVIOR OF WEB SERVICES

One of the important ingredients in a compatibility framework is the *timed conversational protocol* of Web services. In our framework, a timed conversational protocol specifies the sequences of messages a service supports, the involved data flow and the associated timed properties to exchange messages. A timed conversational protocol can be extracted from standard specifications like for example OWL-S (Guermouche and al. 2008-e).

The model we consider is based on *deterministic timed automata*. In fact, this formalism is easy to understand, and at the same time, it is expressive enough to model the aspects and properties that we consider. In addition, several problems have been proven decidable for deterministic timed automata such as complementarity, equivalence, and inclusion (Alur and al. 1994)

Intuitively, the states represent the different phases a service may go through during its interaction. Transitions enable sending or receiving a message. An output message is denoted by *!m*, whilst an input one is denoted by *?m*. A message involving a list of data types is denoted by $m(d_1,...,d_n)$, or $m(\bar{d})$ for short. To capture timed properties when modeling Web services, we propose to use the standard timed automata clocks (Alur and al. 1994). These automata are equipped with a set of clocks. The values of these clocks increase with the passing of time. Transitions are labeled by timed constraints, called *guards*, and resets of clocks. The former represent simple conditions over clocks, and the latter are used to reset values of certain clocks to zero. The guards specify that a transition can be fired if the corresponding guards are satisfied.

A timed constraint is a conjunction of atomic formula that compares the value of a clock $x \in X$, to a positive real constant $a \in R_{\geq 0}$.

Let X be a set of clocks. The set of *constraints* over X, denoted $\Psi(X)$, is defined as follows:

true|x : $a \mid \psi_1 \wedge \psi_2$, where : $\in \{\leq, <, =, \neq, >, \geq\}$, $x \in X$, $\psi_1, \psi_2 \in \Psi(X)$ and *a* is a constant.

We note that the constraints over data are also defined as conjunction of atomic formula that compares the value of a data $d \in D$ to a constant that can be an integer, a real, a string, or a Boolean.

Definition 1 (*Timed conversational protocol*)

A *timed conversational protocol of a Web service* Q is a tuple (S,s_0,F,M,C,X,T) such that:

- S is a set of states,
- s_0 is the initial state ($s_0 \in S$),
- F is the set of final states ($F \subseteq S$),
- M is a set of messages,
- C is the set of constraints over data,
- X is the set of clocks,
- T is a set of transitions such that $T \subseteq S \times M \times C \times \Psi(X) \times 2^X \times S$. A transition $(s, \alpha, c, \psi, Y, s')$ specifies that, from a state s, the service exchanges a message that involves data ($\alpha = ?m(\bar{d})$: input message, $\alpha = !m(\bar{d})$: output message), so that the constraints over data *c* and the temporal constraints ψ are verified. When the transition is fired, clocks *Y* can be reset.

The conversational protocols we consider are deterministic. A conversational protocol is said to be deterministic if for each two transitions $(s, \alpha_1, c_1, \psi_1, Y_1, s_1')$ and $(s, \alpha_2, c_2, \psi_2, Y_2, s_2')$, the following conditions are satisfied :

- $m_1(\bar{d}) \neq m_2(\bar{d})$, or
- $c_1 \wedge c_2 = false$, or
- $\psi_1 \wedge \psi_2 = false$

The set of Web services are equipped with a bounded queue to store the incoming messages.

Example 1.

Figure 2 shows the timed conversational protocols of the services introduced previously. On this figure, the service PS has the set of states $\{p_0,p_1,p_2,p_3,p_4,p_5,p_6,p_7,p_8,p_9,p_{10},p_{11},p_{12},p_{13},p_{14},p_{15},p_{16}\}$. The initial state is p_0, and the final states are p_8 and p_{16}.
This service can send, for example, the message *examination_request(sn, handicap),* denoted *examination_request(sn,handicap)*. This message involves as data types, a *security number* (*sn*) and the *handicap* of the citizen (*handicap*). Analogously, this service can receive messages, for example the message *pension_request(sn,age,handicap)*, denoted *?request-pension(sn,handicap)*. PS terminates correctly its interaction when it is reaches its final state.

To specify that the prefecture sends its final decision after 48 hours and within 96 hours from receiving the pension request, we associate a reset of a clock $t_1=0$ to the transition that enables to receive the request of the pension and we associate the constraint $48 \leq t_1 \leq 96$ to the transition that enables to send the final decision.

In the following sections, we present the key elements to define the semantic of our model.

Clock valuation

The semantic of timed conversational protocols is based on the notion of clock valuation *v*. It associates to each clock *x* a real positive value $R_{\geq 0}$. A clock can be reset, denoted as *v(x) =0*. We say that a clock valuation v(x) satisfies a constraint ψ =x~a, denoted as $v \models \psi$, if v(x)~a (i.e., when we replace the clock *x* by its value, the constraint is satisfied).

Definition 2 *(Clock valuation)*

A valuation of the clocks of X is a function v: $X \rightarrow R_+$. Given a real $\rho \in R_+$, we note $v+\rho$ the valuation that associates to a clock x the value v(x) $+\rho$. If Y is a subset of X, $[Y \leftarrow 0]$v represents the valuation *v′* defined by: v'(x) =0 for each x∈Y. Analogously, the data valuation *u* is a mapping u: $D \rightarrow V_D$ from data to values. The initial valuation u_0 denotes the initial data valuation, such that ∀ d∈D, u_0(d)=null. In addition, at the initial configuration, the queues of the services are empty. We say that a data valuation u(d) satisfies a constraint c=d~b, denoted u ⊨c, if u(d)~b (i.e., when the data d is replaced by its value, the constraint is satisfied). The value of data changes via operations of services.

Timed conversations

By using the clock (resp. data) valuation notion, we define the concept of timed conversations of conversational protocols, inspired from the notion of timed words of timed automata (Alur and al. 1994).

Let Q=(S,s_0,F,M,C,X,T) be a timed conversational protocol. An execution of Q is a sequence of pairs $s_0.(m_0(\bar{d}),t_0).s_1 \ldots s_{n-1}(m_{n-1}(\bar{d}),t_{n-1}).s_n$ such that s_0 is the initial state and s_n is a final state. This execution is said to be correct if when the time increases, the constraints over data and timed constraints are satisfied.

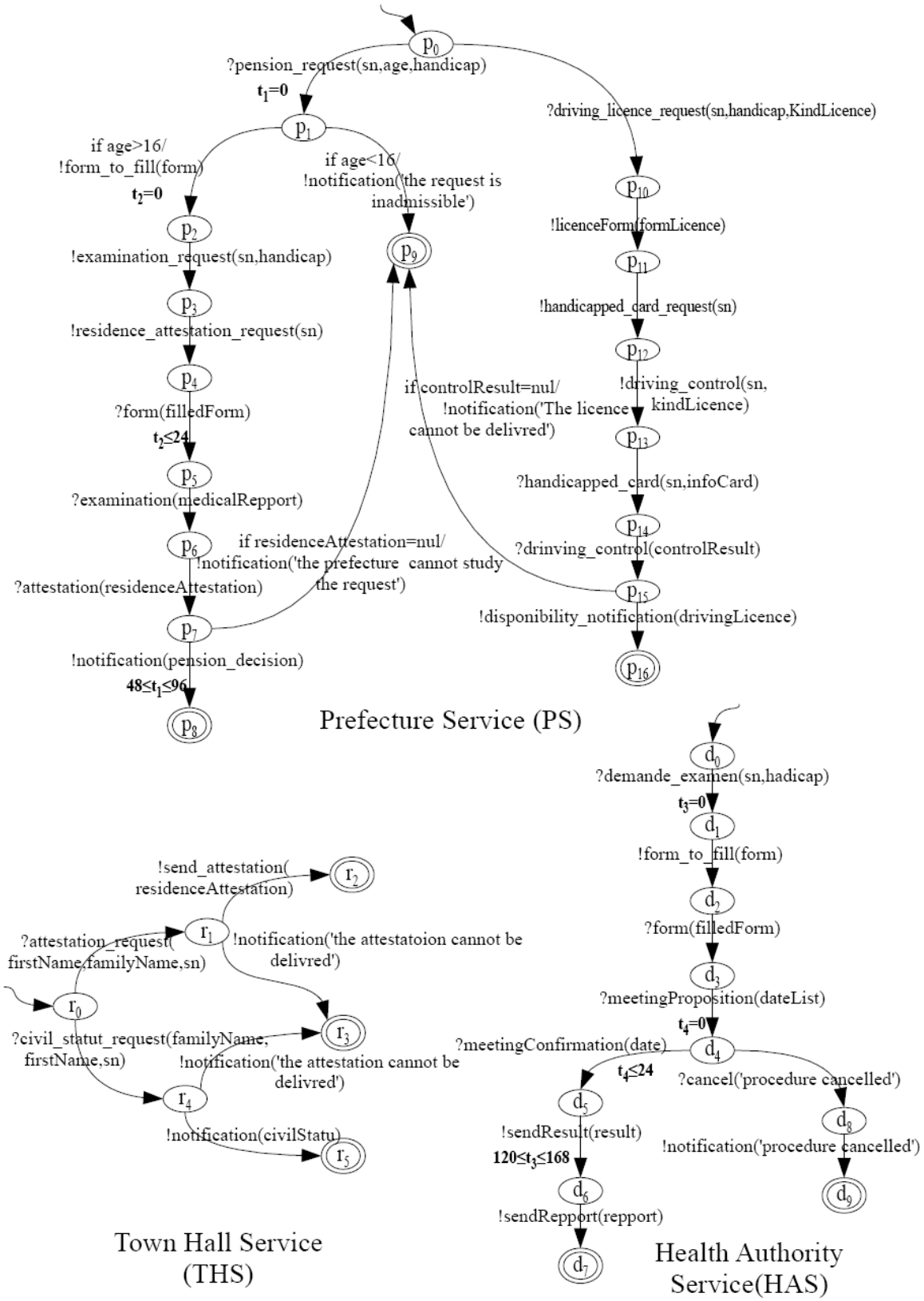

Figure 2. Web services of the e-government scenario

A conversation is the set of observable messages that can be exchanged between services (Ponge. 2008).

Definition 3 *(Correct timed execution)*

Let Q=(S,s_0,F,M,C,X,T) be a timed conversational protocol. An execution which is a sequence $s_0.(m_0(\bar{d}),t_0).s_1 \ldots s_{n-1}(m_{n-1}(\bar{d}),t_{n-1}).s_n$ is said to be correct if:

- $t_0 \leq t_1 \leq \ldots \leq t_n$
- s_0 is the initial state and s_n is the final state
- $\forall i \in [1,n]$, $(s_{i-1},m_{i-1},c_{i-1}, \psi_{i-1},Y_{i-1},s_i)$, $v_{i-1} \models \psi_{i-1}$ and $u_{i-1} \models c_{i-1}$

A correct conversation is defined by the sequence $(m_0(\bar{d}),t_0)\ldots(m_{n-1}(\bar{d}),t_n)$

For example, a correct conversation of the service THS is (?attestation_request(firstName, familyName,sn),0).(!send_attestation(residenceAttestation),10). Such a sequence is called a *timed conversation*. The set of timed conversations constitutes a *timed conversational protocol*.

Semantic of timed conversational protocols

The semantic of timed conversational protocols is defined using a transition relation over configurations made of a state, a clock and data valuation. A service remains in the same state s without triggering a transition when the time increments, if there is no transition $(s, \alpha, c, \psi_X, Y, s')$ such that the constraints over data *c* and the timed constraints ψ_X are satisfied, where $\psi_X \subseteq \Psi(X)$ and α is either an output message $!m(\bar{d})$ or an input message $?m(\bar{d})$ which is available in the queue. In an asynchronous communication, when a message is sent, it is inserted in a message queue, and the receiver consumes (i.e. receives) the message while it is available in the queue.

Definition 4 (*Semantic of timed conversational protocol of an asynchronous service*)

Let P = (S,s_0,F,M,C,X,T) be a conversational protocol and Que the associated empty queue. The semantic is defined as a labeled transition $(\Gamma, \gamma_0, \rightarrow)$, where $\Gamma \subseteq S \times V_T \times U_D$ is the set of configurations, such that V_T is a set of timed valuations, U_D is the set of data valuation, $\gamma_0 = (s_0,u_0,v_0)$ is the initial configuration, and $\rightarrow$ is defined as follows:

- Elapse of time: $(s,u,v) \xrightarrow{tick} (s,u,v+\rho)$
- Location switch: $(s,u,v) \xrightarrow{\alpha} (s',u',v')$, if $\exists t = (s, \alpha, c, \psi_X, Y, s')$ such that $u \models c$ and $v \models \psi$ and $\forall y \in Y$, $v'(y)=0$, $\forall x \in X \setminus Y$, $v'(x) = v(x)$, where $Y \subseteq X$, and
  - If $\alpha = !m(\bar{d})$ then Que:=Que+$m(\bar{d})$
  - If $\alpha = ?m(\bar{d})$ and $m(\bar{d}) \in$ Que then Que:=Que-$m(\bar{d})$

Example 2.

When the PS service, shown in Figure 2, reaches its state p_7 and the value of the clock t_1 is equal to 30, i.e., less than 48 hours, then the service remains on the state p_7 while the time increases. When the value of the clock t_1 is equal or bigger than 48 and less or equal to 96 hours, then the transition $(p_7,!notification(pension\_decision,48),48 \leq t_1 \leq 96,p_8)$ is fired. When this happens, some

calculation can be fulfilled and some data are updated. As the transition allows to send the message *notification(pension_decision)*, then it will be added into the queue of the receiver service.

Next, we will present the intuition behind the choreography compatibility problem.

TIMED COMPATIBILITY PROBLEM

In this section, we discuss informally and intuitively by using examples the timed choreography compatibility problem and the related issues. Since the communication time is very small, we neglect it in our framework.

Example 3.

Let us first consider the two untimed conversational protocols respectively of Q and Q' services depicted in Figure 3. In spite that both services do not produce and consume their messages in the same order, the two asynchronous services are *fully compatible*. The service Q starts by sending the message $m_0(d_0,d_1)$ which becomes available in the queue of Q'. On the other side, Q' sends the message $m_2(d_3)$. After that, Q' consumes the message $m_0(d_0,d_1)$ and then it sends the message $m_1(d_2)$ which is added to the queue of Q. Therefore, Q can consume the message $m_1(d_2)$ and then the message $m_2(d_3)$. Using the existing work, these two services will be considered as incompatible although they can succeed an interleaved execution.

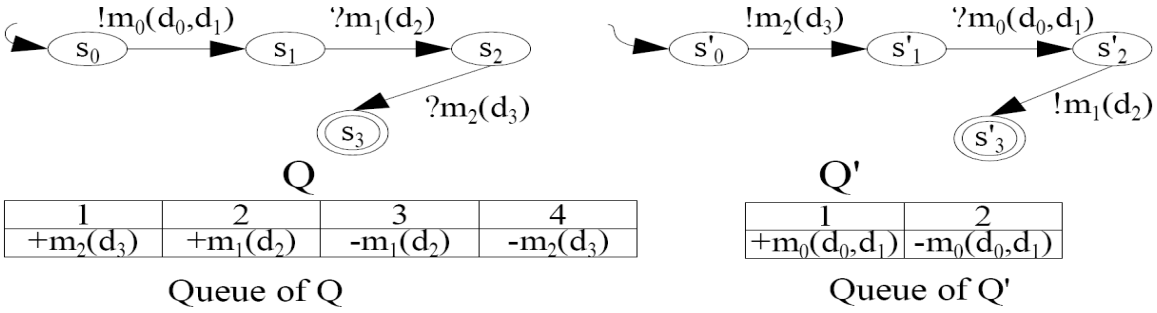

Figure 3. Untimed compatible asynchronous Web services

Augmenting the conversational protocols of asynchronous services by timed properties lays important challenges. Particularly, the clocks used to define timed properties are local and are mutually independent. At the same time, in our work, we do not assume that timed properties are synchronized over messages, i.e., the timed constraints of the different services are not defined over clocks which are necessarily reset at the same time. Consequently, when the services interact together, implicit timed conflicts can arise. To illustrate this issue, let us consider the following example.

With the two timed conversational protocols of the Q and Q' services depicted in Figure 4. The service Q starts by sending the message $m_0(d_0,d_1)$. So this latter becomes available in the queue of Q'. On the other hand, Q' can send the message $m_2(d_3)$ that can be stored in the queue of Q. The service Q remains blocked, since the message $m_1(d_2)$ is not yet available. But Q' can consume the message $m_0(d_0,d_1)$ which has been already sent by Q. Once consumed, Q' sends the message $m_1(d_2)$ after 20 and within 40 units of time from consuming the message $m_0(d_0,d_1)$. Consequently,

the message $m_1(d_2)$ becomes available in the queue of Q after 20 units of time from consuming the message $m_0(d_0,d_1)$. In that case, Q will be able to consume the message $m_1(d_2)$ after 20 units of time. Finally, Q must consume the message $m_2(d_3)$ within 10 units of time. However, this message can be consumed after consuming the message $m_1(d_2)$, i.e., after 20 units of time. In fact, the message $m_1(d_2)$ can be sent (becomes available) by Q' after 20 units of time. So, the message $m_2(d_3)$ must be consumed within 10 units of timed but it is possible to consume it only after 20 units of time: these two constraints are conflicting.

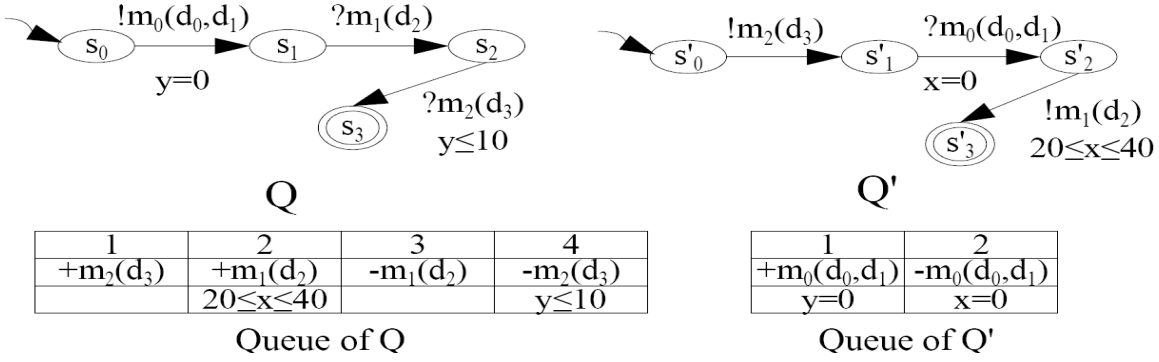

Figure 4. Incompatible timed asynchronous services

We present now another example in which we show another kind of timed conflict.

Example 4

Let us consider the two conversational protocols of the two services P and S depicted in Figure 5. The service P starts by sending the message $m_0(d_0)$ to the service S. Then, the service S sends the message $m_1(d_1)$ the service P must receive within 10 units of time. When the service S sends the message $m_1(d_1)$, the clock x is reset and the clock value of y must be at most 10 units of time. So, the difference between the two clocks x and y must be at most 10 units of time (we suppose that the time of communication is negligible). After that, the service P sends the message $m_2(d_1,d_0)$ after 20 units of time from sending the message $m_0(d_0)$. The service S must receive the message $m_2(d_1,d_0)$ within 5 units of time from sending the message m_1. As said previously, the difference between the two clocks x and y must be at most 10 units of time. However, when the two services exchange the message $m_2(d_1,d_0)$, the value of the clock x must be at most 5 units of time ($x \leq 5$) and the value of the clock y must be at least 20 units of time ($y \geq 20$). According to these values, the difference between the two clocks x and y can never be at most 10 units of time.

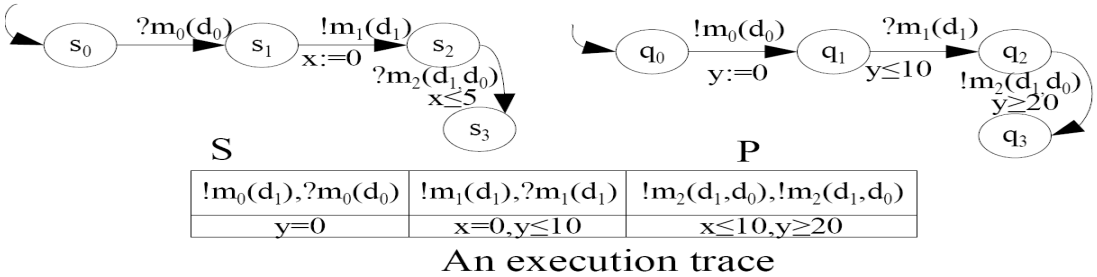

Figure 5. Timed incompatibility due to the difference between the clocks values

This kind timed incompatibility, such as the incompatibility due to the difference between the clocks, cannot be detected by existing works on the compatibility of Web services.

Compatibility classes

In choreography, heterogeneity of services can have a partial or a total impact on their collaboration. Based on the kind of the impact of the heterogeneity, we consider three general classes: (1) *full compatibility*, (2) *partial compatibility*, and (3) *full incompatibility*. The full and partial compatibility classes gather two sub-classes: (a) *perfect compatibility* and (b) *non-perfect compatibility*.

Full compatibility

This first class is assigned to a set of Web services that can collaborate without an eventual deadlock. For example, the two services illustrated in Figure. 6 are fully compatible. In fact, the two services can exchange the two following conversations in which no deadlock arises:

- (($!m_0(d_1)$, $t_1=0$) .$!m_3(d_0)$. ($!m_2(d_2,d_1)$, $t_2=0$) . ($?m_2(d_2,d_1)$, $168 \leq t_1 \leq 336$). ($?m_0(d_1)$, $t_2 \leq 336$) . $?m_3(d_0)$)
- (($!m_3(d_1)$, $t_2=0$) . $!m_4(d_1)$. ($!m_5(d_1)$, $t_2 \leq 24$) . $?m_5(d_1)$.($?m_4(d_1)$, $t_2=0$) . ($?m_3$, $t_2 \leq 48$))

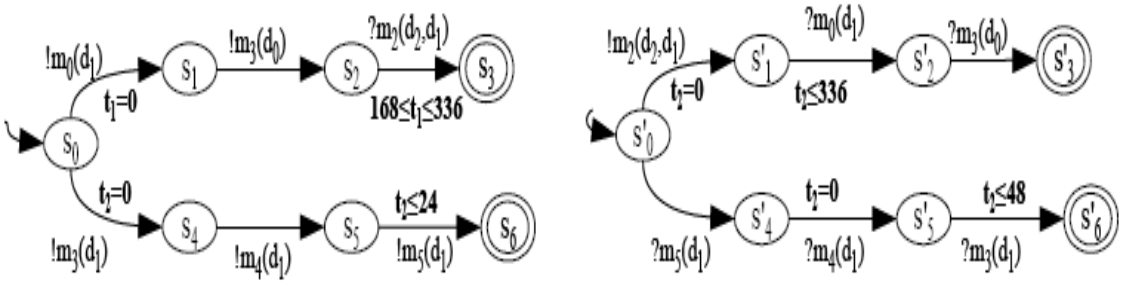

Figure 6. Full compatibility

Since we deal with asynchronous communicating services, the messages can be sent without synchronization, i.e., during an interaction, output messages can be not consumed. For this reason, we distinguish two subclasses: (1) *full and perfect compatibility* and (2) *full but non-perfect compatibility*.

1. *Perfect and full compatibility*: A set of Web services is said to be perfectly and fully compatible, if all their interactions do not contain deadlocks and at the same time, all the produced messages are consumed. For example, the two services illustrated in Figure. 6 are perfectly and fully compatible.

2. *Full but non-perfect compatibility*: The full but non-perfect compatibility concerns services that can interact correctly without deadlocks but at the same time, during their interactions, there are messages that are produced but not consumed, i.e., there are extra messages. For example, the two services illustrated in Figure. 7 are fully but not-perfectly

compatible. In fact, the service Q can send the extra message $m_3(d_4)$ that will not be consumed by Q'. So when the services are interacting together, there are no deadlocks but there is at least one extra (unuseful) message. This case is called *full but non-perfect compatibility*.

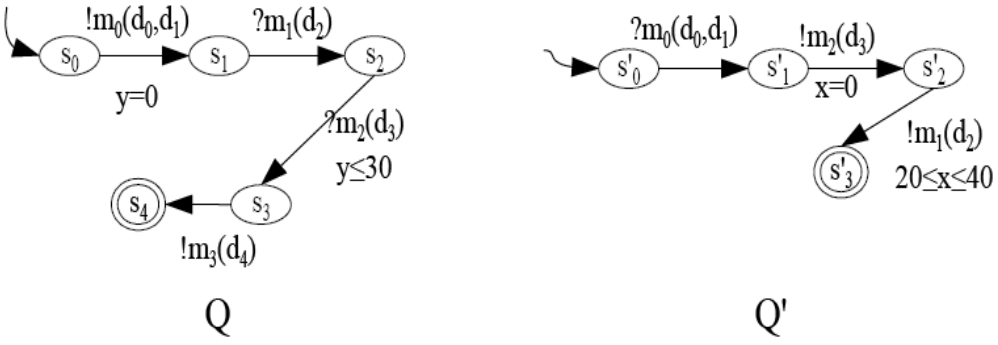

Figure 7. Full but non-perfect compatibility

Partial compatibility

This second class is assigned to a set of services that can partially collaborate correctly. Some interactions of services can be incompatible while other interactions can be correct. As we can see in Figure. 8, the two services can perform correctly the conversation:

$((!m_0(d_1),t_1=0),!m_3(d_0),(!m_2(d_2,d_1),t_2=0),(?m_0(d_1),t_2 \leq 336),(?m_2(d_2,d_1),168 \leq t_1 \leq 336),?m_3(d_0))$

However, the conversation

$((!m_5(d_1),(!m_3(d_5),t_1=0),(?m_3(d_5),t_2=0),(!m_4(d_2),t_2 \geq 48),?m_4(d_2),(?m_5(d_1),t_1 \leq 24))$

is not correct, since the message $m_5(d_1)$ must be consumed within 24 units of time from sending the message $m_3(d_5)$ and at the same time $m_5(d_1)$ must be consumed after consuming $m_4(d_2)$, i.e., after 48 units of time. That means that the message $m_5(d_1)$ must be consumed so that $48 \leq t_1 \leq 24$, which represents a timed deadlock. Since the two services can achieve correctly at least one interaction, and fail at least one interaction, we say that the two services are *partially compatible.*

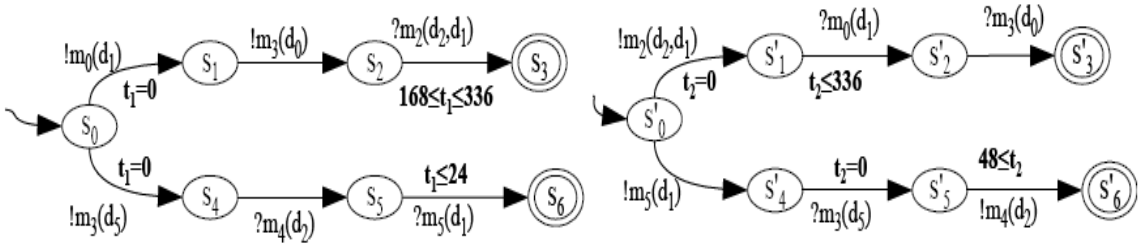

Figure 8. Partial compatibility

Like the class of full compatibility, we distinguish two subclasses of partial compatibility: (1) *perfect partial compatibility* and (2) *not-perfect partial compatibility*.

1. *Perfect partial compatibility*: A set of Web services is said to be perfectly and partially compatible, if they fail at least one conversation and at the same time, during all the correct conversations, all the produced messages are consumed.

2. *Not-perfect partial compatibility*: When a set of Web services are partially compatible and at the same time, there exists at least one correct interaction during which there is at least one extra message, we say that the services are partially but non-perfectly compatible. For example, the two services depicted in Figure 9, fail the conversation

   $$(!m_5(d_1),(!m_3(d_5),t_1=0),(?m_3(d_5),t_2=0),(!m_4(d_2),t_2 \geq 48),?m_4(d_2),(?m_5(d_1),t_1 \leq 24))$$

   but at the same time, they succeed the conversation

   $((!m_0(d_1),t_1=0),!m_3(d_0),(!m_2(d_2,d_1),t_2=0),(?m_0(d_1),t_2 \leq 336),(?m_2(d_2,d_1),168 \leq t_1 \leq 336),!m_5(d_0),$ $?m_3(d_0))$

   But, in this conversation, the message $m_5(d_0)$ is a message that will not be consumed, i.e., it is an extra message.

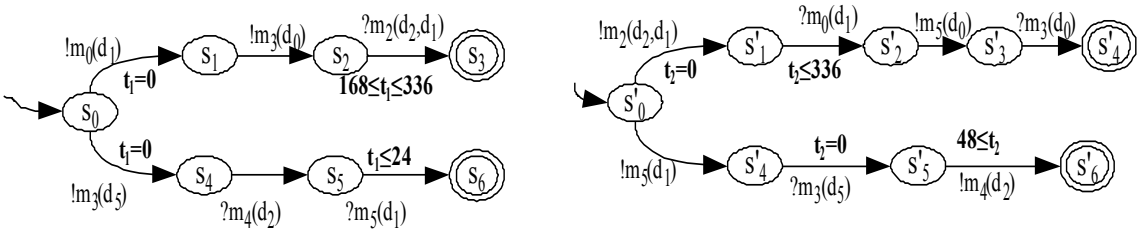

Figure 9. Partial but non-perfect compatibility

Full incompatibility

This class of compatibility is attributed to a set of Web services for which a deadlock arises in all their interactions. For example, the services illustrated in Figure. 10 fail all their interactions. In fact, Q' sends the message $m_1(d_2,d_1)$ and Q consumes it. Then, the two services remain blocked while waiting respectively the messages $m_2(d_0)$ and $m_3(d_1)$.

In the second conversation, the same problem described in the previous section arises. As the two services fail all their conversations, so we say that they are fully incompatible.

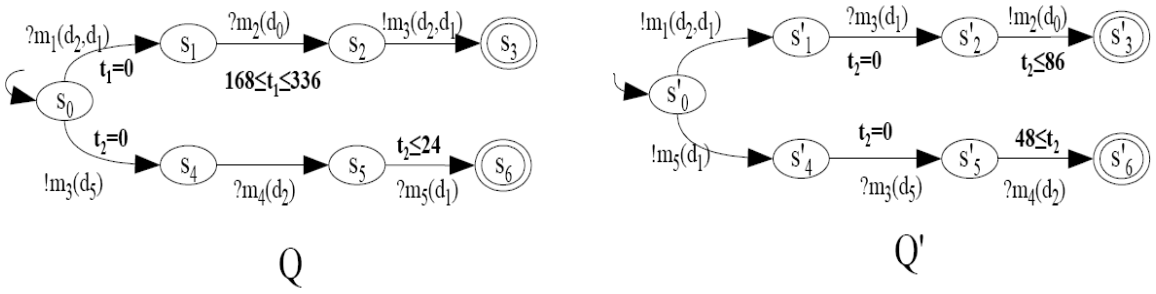

Figure 10. Full incompatibility

In order to handle the eventual timed conflicts, we propose an UPPAAL model checker based approach. To do so, we first propose some transformation to handle *timed asynchronous services*.

FROM CONVERSATIONAL PROTOCOLS TO UPPAAL TIMED AUTOMATA

UPPAAL is a model checker for the verification and simulation of real time systems. An UPPAAL model is a set of timed automata, clocks, channels for systems (automata) synchronization, variables and additional elements (Larsen and al. 1997).

Each automaton has one initial state. Synchronization between different processes can take place using channels. A channel can be written into (denoted as channel_name!), and can be read (denoted as channel_name?). A channel can be defined as urgent to specify that the corresponding transition must be fired as soon as possible, i.e. immediately and without a delay.
The conditions associated to transitions, called guards, specify that a transition can be fired if the corresponding guards are satisfiable. The conditions associated to states, called invariants, specify that the system can stay in the state while the invariant is satisfiable. Variables and clocks can be associated to processes (automaton). Conditions on these clocks and variables can be associated to transitions and states of the process.

The UPPAAL properties query language is a subset of *Computation Tree Logic* (CTL) (Henzinger and al. 1994). The properties that can be analyzed by UPPAAL are:

- $\forall\Box\psi$: for all the automata' paths, the property ψ is always satisfiable, i.e., for each transition (or a state) of each path, the property ψ is satisfiable.
- $\forall\Diamond\psi$: for all the automata' paths, the property ψ is eventually satisfiable, i.e., for each path , there is at least one transition (or a state) in which the property ψ is satisfiable.
- $\exists\Box\psi$: there is at least a path in the automata such that the property ψ is always satisfiable, i.e., there is at least one path such that for each transition (or a state), the property ψ is satisfiable.
- $\exists\Diamond\psi$: there is at least a path in the automata such that the property ψ is eventually satisfiable, i.e., there is at least one transition (or a state) of at least one path in which the property ψ is satisfiable.
- $\psi \rightarrow \varphi$: when ψ holds, φ must hold.

In order to perform a model checking by using UPPAAL, we propose a set of transformation steps, which are:

- Messages abstraction
- Data constraints abstraction
- Invariant association
- Urgent channel association
- Specifying final states

Abstraction of messages by variables

As mentioned above, UPPAAL holds the notion of channels to synchronize real systems. By using such property, only synchronous services can be analyzed (Guermouche and al. 2008-d). As our framework deals with asynchronous services, hence, the use of the notion of channels is inadequate. To succeed the verification of choreography compatibility of asynchronous services, we propose the idea of messages abstraction that correspond to the notion of channels in UPPAAL.

To do so, we propose the mapping of messages to *variables*. The idea is to abstract each message by a variable whose initial value is zero. We map the common messages (i.e., the same signature) of the conversational protocols to the same variable. To do so, we first compute the common set of messages of the timed conversational protocols. The purpose of the former is to abstract (represent) all the messages that have the same signature, i.e., the same name and same data by the same variable. For example, if one service can send (resp. receive) the message $m_1(d_1,d_2)$ and the other service can receive (resp. send) these message $m_1(d_1,d_2)$, we abstract the messages by using, for example, the variable m_1.

Abstraction by incrementing and decrementing variables

The services are equipped with a queue to store the incoming message. When a message is sent (resp. consumed), it will be inserted to (resp. removed from) the queue. So in this case the occurrences number of the message is incremented (resp. decremented) by one. In order to simulate the queue transactions, we propose to represent the notion of output (resp. input) message by an incrementing operation (resp. decrementing) of the variables value associated to messages.
When a message is sent, we increment the value of the associated variable by one. When this message is received, we decrement the value of the associated variable by one. Note that, a message can be consumed if it is available in the queue. This latter is equivalent to check if the value of the corresponding variable is not equal to zero. So a transition that enables an input message can be fired if the corresponding variable is bigger than zero. We associate to such transitions the constraint that allows to check if the value of the associated variable is bigger than zero.

Definition 5. *(Messages abstraction)*

Let $Q_1=(S_1,s_{01},M_1,C_n,X_1,T_1)$, …, $Q_n=(S_n,s_{0n},M_n,C_n,X_n,T_n)$ be n conversational protocols and R be a set of variables that have zero as initial value. We define the abstraction function Abs: $M \rightarrow R$ that maps messages to variables. For each $m(\overline{d}) \in \bigcup_{i=1}^{n} M_i$ and $r \in R$, $m(\overline{d}) \rightarrow r$ is defined as :

- $(s_i,!m(\bar{d}),c,\psi,Y,s_i') \rightarrow (s_i,r{+}{+},c,\psi,Y,s_i')$
- $(s_i,?m(\bar{d}),c,\psi,Y,s_i') \rightarrow (s_i,r{-}{-},r{>}0,c,\psi,Y,s_i')$

Example 5

As we can see in Figure 11, the set of message of the two services Q and Q' is $\{m_0(d_1),m_3(d_0),m_2(d_2,d_1),m_1(d_1),m_2(d_3)\}$. We abstract respectively each message by a variable $\{m_0, m_3, m_{21}, m_1, m_{22}\}$. The transition $(s_0,!m_0(d_1),s_1)$ of Q enables to send the message $m_0(d_1)$. Once this message is sent, it will be added to the queue of Q'. So, we represent the sending operation of messages by incrementing operation of the corresponding variable. So the transition $(s_0,!m_0(d_1),s_1)$ will be represented by the transition $(s_0,m_0{+}{+},s_1)$. We apply this step to all the transitions that enable output messages.

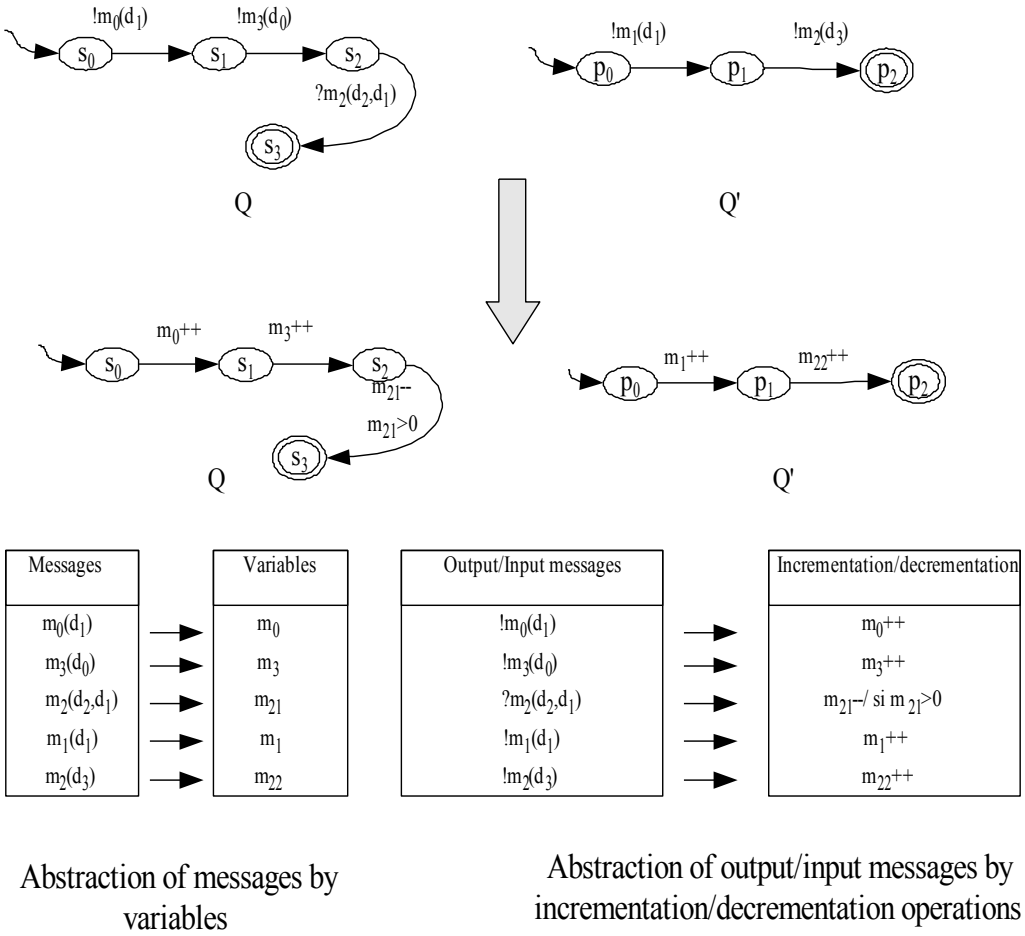

Figure 11. Abstraction of messages

When a transition enables an input message, we propose to represent it by decrementing the corresponding variable. For example, when the transition $(s_2,?m_2(d_2,d_1),s_3)$ is fired, the message $m_2(d_2,d_1)$ will be consumed. So, we abstract the message $?m_2(d_2,d_1)$ by $m_2{-}{-}$. On other side, this transition can be fired if the message is available in the queue. This condition is equivalent to check that the value of the corresponding variable is bigger than zero. By applying this step, the transition $(s_2,?m_2(d_2,d_1),s_3)$ will be represented by the transition $(s_2,m_2{-}{-},m_2{>}0,s_3)$.

Abstraction of data constraints

As said previously, our model considers constraints over data. These constraints can be specified as constraints over non-timed variable. To analyze these constraints by UPPAAL, the values of the variables must be known. However, as the compatibility analysis we propose is done at design time, the values of the variables cannot be known in advance. Consequently, the constraints over data cannot be correctly considered. To consider constraints over data, we propose to abstract again the variables of messages resulting from the process of abstraction of messages described above following the constraints of data. The idea is to compute the set of transitions that hold the same variable. If the set of solutions of the data constraints associated to these transitions is disjoined, then we abstract differently the variables. Whilst, if the set of solution is not disjoined, then we remove only the data constraints without changing the variables. To explain this issue, let us present the following example.

Example 6

Via this example, we will show how we apply the data constraints abstraction process. As we can see, the two services, illustrated in Figure 12, have three common transitions (i.e., transitions that hold the same variable):

- $(s_0,m_0{+}{+},d_0<100,s_1)$ and $(p_1,m_0{+}{+},d_0>120,p_2)$
- $(s_1,m_3{+}{+},d_3>10,s_2)$ and $(p_2,m_3{-}{-}, m_3>0,d_2<10,p_3)$
- $(s_2,m_{21}{-}{-}, m_{21}>0,d_1<50,s_3)$ and $(p_0,m_{21}{+}{+},d_1<80,p_1)$

Let us start by the first pair of transitions $(s_0,m_0{+}{+},d_0<100,s_1)$ and $(p_1m_0{+}{+},d_0>120,p_2)$. We can remark that the set of solutions of the constraints $d_0<100$ and $d_0>120$ is disjoint, i.e., $Sol(d_0<100) \cap Sol(d_0>120)= \varnothing$. Hence, by applying the data constraints abstraction process, we substitute $m_0{+}{+}$ of the transition $(p_1,m_0{+}{+},d_0>120,p_2)$ by another variable m_0'. So the transition becomes $(p_1,m_0'{+}{+},p_2)$

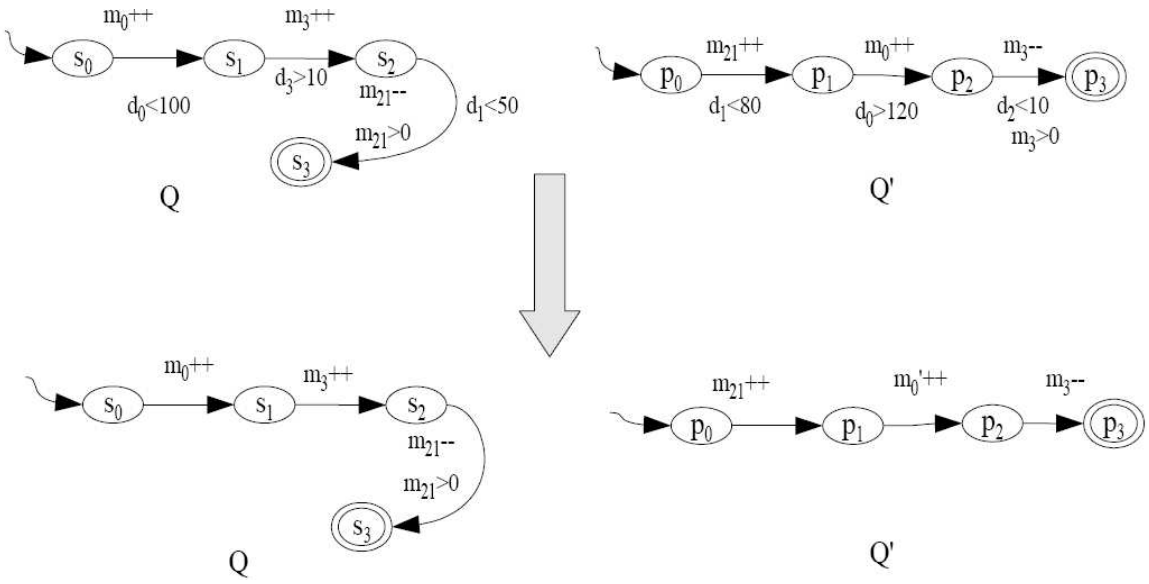

Figure 12. Abstraction of data constraints

Now, we check the second pair of transitions (s_1,m_3++,d_3>10,s_2) and (p_2,m_3--, m_3>0,d_2<10,p_3). The first transition can be fired if the value of the data d_3 is bigger than 10. The second transition can be fired if the value of the variable d_2 is less than 10. Since the data constraints are specified over different data, then when abstracting data constraints, we do not substitute the variable m_3 of the two transitions.

Finally, we verify the last pair of transitions (s_2,m_{21}--, m_{21}>0,d_1<50,s_3) and (p_0,m_{21}++,d_1<80,p_1). We can see that the set of solutions of data constraints d_1<50 and d_1<80 is not disjoint. The two constraints have a common set of solutions, i.e., $Sol(d_1<50) \cap Sol(d_1<80) \neq \varnothing$. Consequently, when abstracting data constraints, we do not substitute the variable m_{21}.

Invariant association

UPPAAL timed automata holds the notion of *invariant*. This latter specifies that the system can stay in the state while the associated invariant is satisfiable. Whilst, in our model, we do not consider invariant. The semantic we define allows to a transition to be fired as soon as possible (once the associated guard is satisfiable). In UPPAAL timed automata semantic, when a transition has a guard of the form $x \geq v$ (resp. x>v) and at the same time the source state of this transition does not have an invariant, the process can stay infinitely in this state. Consequently, such property gives rise to a deadlock. To prevent such setting, we associate for each state source of a transition having a guard of the form $x \geq v$[1] (resp. x>v) an invariant of the form $x \leq v$ (resp. x<v) to constrain the service to stay a limited time in the state.

Example 7

Figure 13 shows a timed conversational protocol to which we associate an invariant.

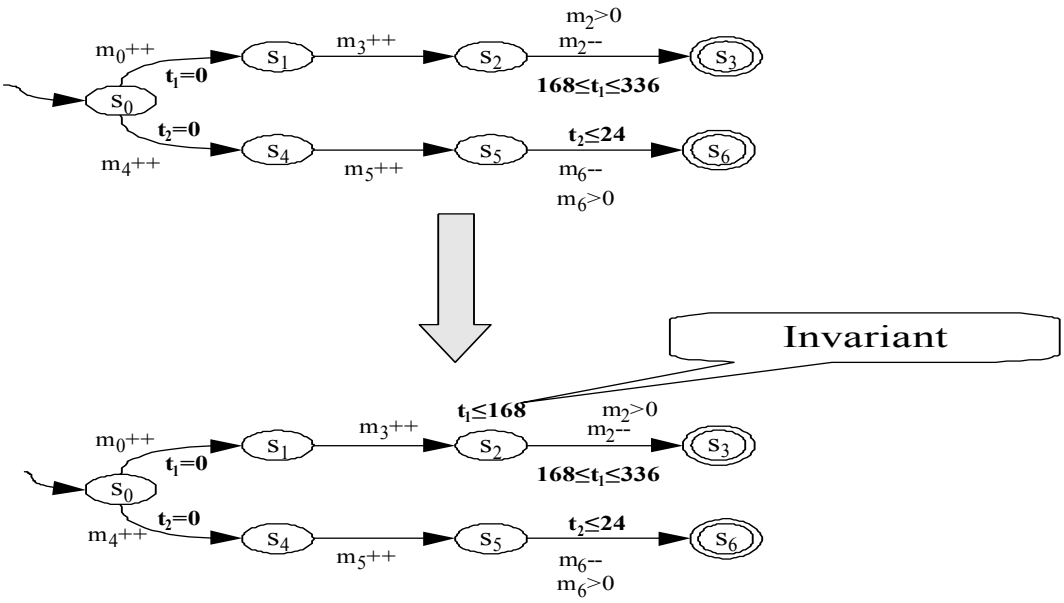

Figure 13. Invariant association

[1] The value used to define the invariant can be any value v' such that $v' \geq v$

When analyzing this protocol with UPPAAL, a deadlock arises at the transition (s_2,m_2--,m_2>0, $168 \leq t_1 \leq 336$, s_3) . In fact, this transition holds the constraint $168 \leq t_1$ and according to the semantic used in UPPAAL, the service can stay infinitely in the state s_2. To constrain the service to leave the state s_2, we associate the invariant $t_1 \leq 168$.

Association of urgent broadcast empty channel to transitions

In the previous section, we have shown how we map the states which are a source state of a transition having a constraint of the form $x \geq v$ (resp. x>v). As in the model we have defined, the transitions of the services must be fired as soon as possible (i.e., when the guards are satisfiable), we associate to the transitions that do not have guards an urgent broadcast output channel. This later aims to simulate the fact that transitions must be fired as soon as possible.

In UPPAAL, a channel *cv* can be defined as urgent as follows:

urgent chan cv;

As the UPPAAL model is based on messages synchronization concept (i.e., a message can be sent only if there is the input counterpart), so to be able to send message without synchronization with an input counterpart, we define the urgent channel as a *broadcast channel*.

urgent broadcast chan cv;

Example 8

In Figure 14, we associate to the transitions that do not hold timed constraints the output urgent channel *cv*. The aim of this is to constrain transitions to be fired as soon as possible.

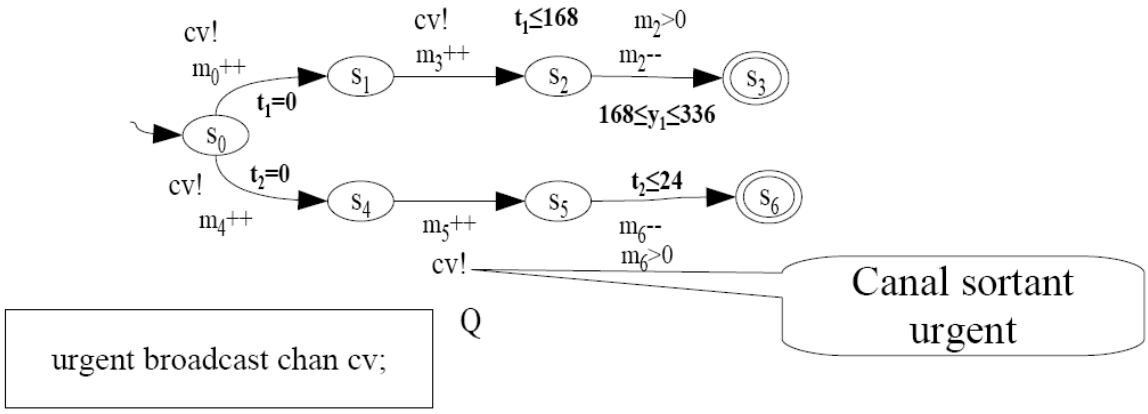

Figure 14. Association of urgent output channel

Defining final states

A correct interaction of a service is the interaction that terminates in a final state. In the standard timed automata handled by UPPAAL, there is no final state notion. As shown in Figure 15, to simulate the final states by UPPAAL timed automata, we define a particular state called *final* that represents the correct termination of the services.

The result of the transformation steps we described above is a set of UPPAAL timed automata. These automata preserve the semantic we consider in timed conversational protocol of asynchronous services.

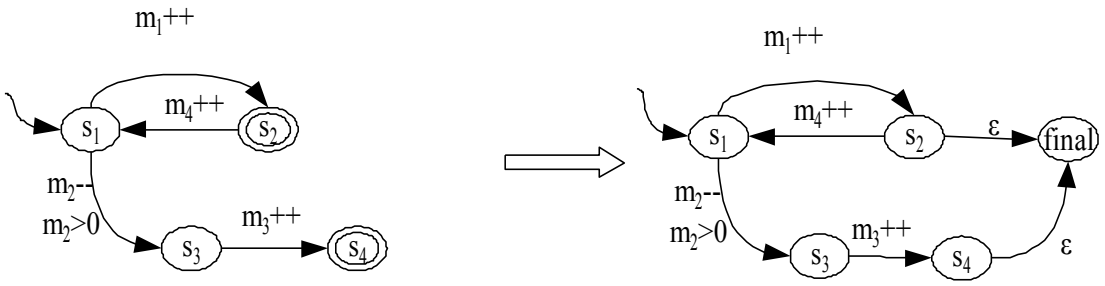

Figure 15. Defining final states in timed automata handled by UPPAAL

Definition 6. *(Transformed timed conversational protocol)*

UPPAAL supported timed automata Q resulting from the transformation process is defined by the tuple $(S,s_0,s_f,cu,R,X,VM,T,Inv)$ such that:

- S is the set of states
- s_0 is the initial state
- s_f is the final state
- cu is the urgent channel
- R is the set of variables used to abstract messages
- X is the set of clocks
- VM is the set of constraints over variables of R
- T is the set of transitions such that: $T \subseteq S \times OP(R) \times VM \times \Psi(X) \times 2^X \times S$ which specifies that when a transition is fired, we perform an operation OP (we increment the variable r++: corresponds to an output message, decrement the variable r--: corresponds to an input message), a constraint over the variables VM (if OP(R)=r--, VM=r>0, else VM=$\varnothing$), a timed constraint, and the set of clocks to reset).
- Inv: $S \rightarrow \Psi(X)$ associates an invariant to states

Figure 16 depicts the timed automata specification of the motivating example depicted in Figure 2 resulting from the abstraction process.

Next, we present the formal primitives we propose to characterize the compatibility class of a set of timed asynchronous Web services.

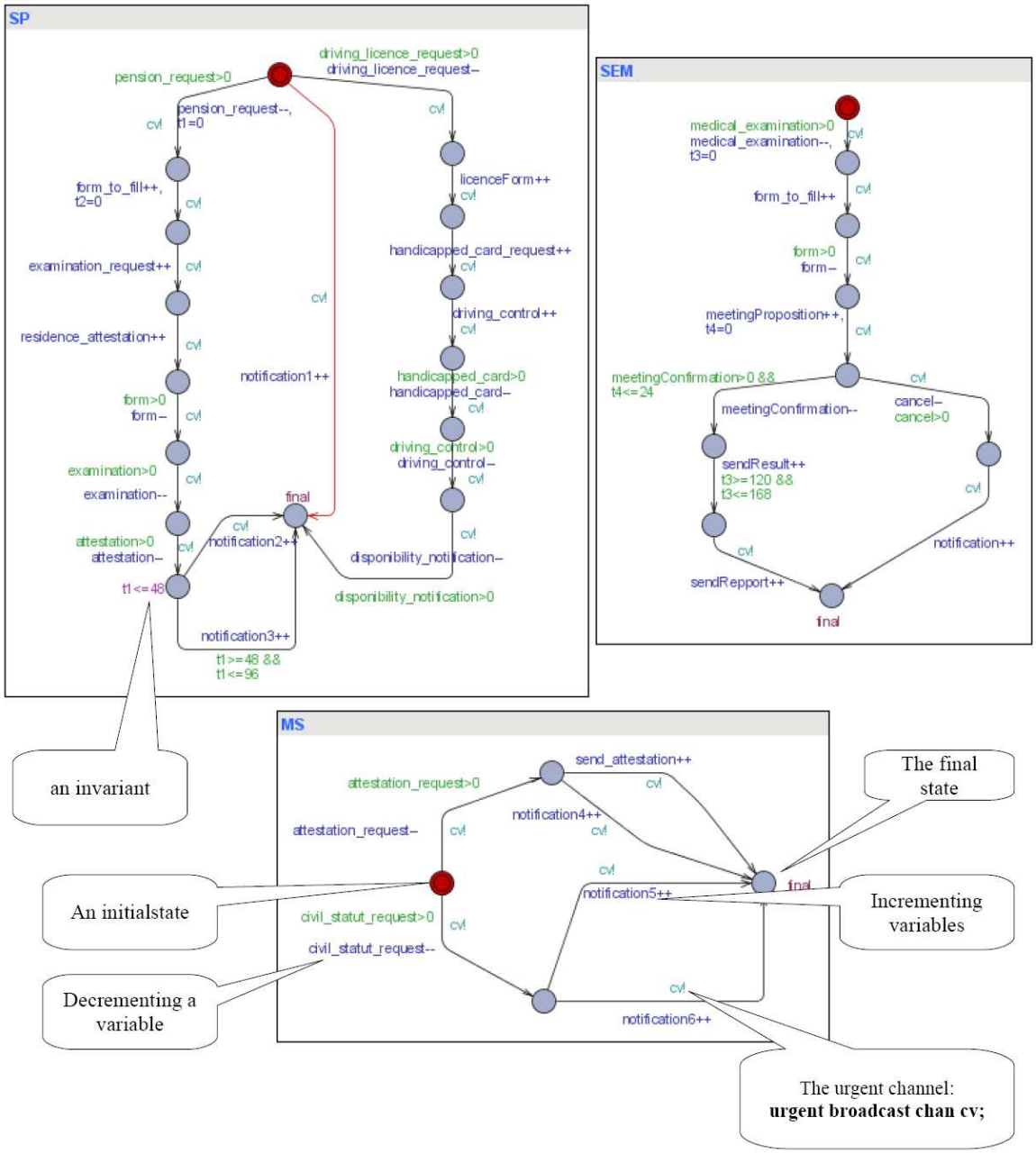

Figure 16. UPPAAL automata resulting from the abstraction process of the e-government scenario services

FORMAL ASYNCHRONOUS COMPATIBILITY CHECKING

In this section, we present compatibility checking with the model checker UPPAAL. We define five compatibility classes: (1) *full and perfect compatibility*, (2) *full but non-perfect compatibility,* (3) *perfect partial compatibility*, (4) *partial but non-perfect compatibility,* and (5) *full incompatibility*.

Perfect full compatibility

In general, a set of Web services constitute a full compatible choreography if they can interact without an eventual blocking. As we deal with asynchronous services, the output messages are sent without synchronization with the corresponding input. As introduced above, it is not sufficient to check only if there is no a deadlock when the services interact together, but, it is important to check that all the sent messages are consumed. So, a set of services constitutes a *full and perfect compatible choreography* if: (1) they collaborate together without any eventual blocking and (2) at the same time, all the generated messages are consumed.

Formally, checking that a set of Web services can interact without an eventual blocking is equivalent to check that the services reach their final states in all interactions. At the same time, when the services reach their final states, the fact that all the sent messages must be consumed is formally equivalent to check that, when the services reach their final states, all the variable values are equal to zero. Remember that, the value of the variables represents the number of the occurrences of the corresponding sent message.

Let $P_1,\ldots,P_n$ be n (asynchronous) services and $R_1,\ldots,R_n$ be the corresponding set of variables. The set of fully and perfect compatible Web services is specified by the following CTL formulas:

$$\forall \Diamond P_1.final \wedge \ldots P_n.final \wedge P_1.final \wedge \ldots \wedge P_n.final \rightarrow r_1==0 \wedge \ldots \wedge r_m==0, \quad (1)$$

Example 9.

According to the motivating example, PS, HAS, and THS services constitute a full and perfect compatible choreography iff:

$$\forall \Diamond PS.final \wedge HAS.final \wedge THS.final \wedge PS.final \wedge HAS.final \wedge THS.final \rightarrow disponibility\_notification == 0 \wedge driving\_control == 0 \wedge handicapped\_card==0 \wedge handicapped\_card\_request==0 \wedge licenceForm==0 \wedge driving\_licence\_reques==0 \wedge driving\_licence\_request==0 \wedge notification2==0 \wedge notification1==0 \wedge notification3==0 \wedge examination==0 \wedge form==0 \wedge residence\_attestation==0 \wedge examination\_request==0 \wedge attestation==0 \wedge form\_to\_fill==0 \wedge pension\_request==0 \wedge send\_attestation==0 \wedge civil\_statut\_request==0 \wedge attestation\_request== \wedge sendRepport==0 \wedge notification6==0 \wedge notification5==0 \wedge notification4==0 \wedge notification==0 \wedge sendResult==0 \wedge cancel==0 \wedge meetingConfirmation==0 \wedge meetingProposition==0 \wedge medical\_examination==0$$

Non-perfect full compatibility

When a set of Web services can collaborate together without an eventual blocking but at the same time, during their interaction, there are messages that cannot be consumed, i.e., extra messages, we say that the services are *fully but non-perfectly compatible*.

Formally, a set of Web services are said to be *fully and non-perfectly compatible* if via all the paths, the services reach their final state and at the same time, there exists at least one variable whose value is bigger than zero. This latter can be specified by the following CTL formulas:

$$\forall \Diamond P_1.final \wedge \ldots \wedge P_n.final \wedge \exists\Diamond (P_1.final \wedge \ldots \wedge P_n.final \text{ imply } r_1>0 \vee \ldots \vee r_m>0) \text{ Where } r_i \in \{R_1,\ldots,R_n\} \quad (2)$$

Example 10.

The three services PS, HAS, and THS are said to be fully but non-perfectly compatible iff:

$\forall \Diamond$ PS.final $\wedge$ HAS.final $\wedge$ THS.final $\wedge$
$\exists\Diamond$ (PS.final $\wedge$ HAS.final $\wedge$ THS.final $\rightarrow$ disponibility_notification>0 $\vee$ driving_control >0 $\vee$ handicapped_card>0 $\vee$ handicapped_card_request>0 $\vee$ licenceForm>0 $\vee$ driving_licence_reques>0 $\vee$ driving_licence_request>0 $\vee$ notification2>0 $\vee$ notification1>0 $\vee$ notification3>0 $\vee$ examination>0 $\vee$ form>0 $\vee$ residence_attestation>0 $\vee$ examination_request>0 $\vee$ attestation>0 $\vee$ form_to_fill>0 $\vee$ pension_request>0 $\vee$ send_attestation>0 $\vee$ civil_statut_request>0 $\vee$ attestation_request>0 $\vee$ sendRepport>0 $\vee$ notification6>0 $\vee$ notification5>0 $\vee$ notification4>0 $\vee$ notification>0 $\vee$ sendResult>0 $\vee$ cancel>0 $\vee$ meetingConfirmation>0 $\vee$ meetingProposition>0 $\vee$ medical_examination>0)

Partial but non-perfect compatibility

As services are heterogeneous they can fulfill incorrect conversations. A conversation during which a service remains blocked is called *incorrect*. A set of Web services are not fully compatible when the set of possible conversations of the services hold at least one incorrect conversation.

Formally, a set of Web services are not fully compatible if there exists at least a path of their automata that cannot reach the final state. This later can be specified as the following formula:

$$\exists\Box\neg P_1.final \vee \ldots \vee \exists\Box\neg P_n.final \quad (3)$$

When a set of Web services can achieve correctly a set of conversations and at the same time they fail other conversations, we say that the services are partially compatible. In this section, we define particularly the *partial but non-perfect compatibility* class. This class is assigned to a set of Web services that are partially compatible and at the same time, there is at least one correct conversation during which there is at least one extra message. This is formally specified by the following CTL formulas:

$$\exists\Diamond P_1.final \wedge \ldots \wedge P_n.final \wedge \exists\Diamond (P_1.final \wedge \ldots \wedge P_n.final \rightarrow r_1>0 \vee \ldots \vee r_m>0) \text{ where } r_i \in \{R_1,\ldots,R_i,\ldots,R_n\} \quad (4)$$

Formally, a set of Web services is said to be partially but non-perfectly compatible if the formulas (3) and (4) are satisfied.

Example 11.

The three services PS, HAS, and THS are said to be partially but non-perfectly compatible iff:

$$\exists\Box\neg PS.final \vee \exists\Box\neg HAS.final \vee \exists\Box\neg THS.final \wedge \exists\Diamond PS.final \wedge HAS.final \wedge THS.final \wedge \exists\Diamond (PS.final \wedge HAS.final \wedge THS.final \rightarrow disponibility\_notification>0 \vee driving\_control >0 \vee handicapped\_card>0 \vee handicapped\_card\_request>0 \vee licenceForm>0 \vee driving\_licence\_reques>0 \vee driving\_licence\_request>0 \vee notification2>0 \vee notification1>0 \vee notification3>0 \vee examination>0 \vee form>0 \vee residence\_attestation>0 \vee examination\_request>0 \vee attestation>0 \vee form\_to\_fill>0 \vee pension\_request>0 \vee send\_attestation>0 \vee civil\_statut\_request>0 \vee attestation\_request>0 \vee sendRepport>0 \vee notification6>0 \vee notification5>0 \vee notification4>0 \vee notification>0 \vee sendResult>0 \vee cancel>0 \vee meetingConfirmation>0 \vee meetingProposition>0 \vee medical\_examination>0)$$

Partial and perfect compatibility

The partial and perfect compatibility class characterizes the fact that services are not fully compatible but at the same time, they can fulfill correctly conversations during which all the produced messages are consumed.

Formally, a set of Web services can achieve correctly at least one conversation so that all produced message are consumed is equivalent to check that there exists at least one path so that all the services reach their final state and at the same time, when the final state is reached, the value of all variables is equal to zero. This is specified by the following CTL formula:

$$\exists\Diamond P_1.final \wedge \ldots \wedge P_n.final \wedge \exists\Diamond (P_1.final \wedge \ldots \wedge P_n.final \rightarrow r_1==0 \wedge \ldots \wedge r_m==0) \text{ where } r_i \in \{R_1,\ldots,R_i,\ldots,R_n\} \quad (5)$$

A set of Web services whose conversational protocols do not verify the formula (4) (i.e., the partial but non-perfect compatibility is not verified) and at the same time, verify the formulas (3) and (5) is said to be *partially and perfectly compatible*.

Example 12.

The three services PS, HAS, and THS are said to be partially and perfectly compatible if:

$$\exists\Box\neg\, PS.final \vee \exists\Box\neg\, HAS.final \vee \exists\Box\neg\, THS.final \wedge \exists\Diamond PS.final \wedge HAS.final \wedge THS.final \wedge \exists\Diamond\, (PS.final \wedge HAS.final \wedge THS.final \rightarrow disponibility\_notification==0 \wedge driving\_control ==0 \wedge handicapped\_card==0 \wedge handicapped\_card\_request==0 \wedge licenceForm==0 \wedge driving\_licence\_reques==0 \wedge driving\_licence\_request==0 \wedge notification2==0 \wedge notification1==0 \wedge notification3==0 \wedge examination==0 \wedge form==0 \wedge residence\_attestation==0 \wedge examination\_request==0 \wedge attestation==0 \wedge form\_to\_fill==0 \wedge pension\_request==0 \wedge send\_attestation==0 \wedge civil\_statut\_request==0 \wedge attestation\_request==0 \wedge sendRepport==0 \wedge notification6==0 \wedge notification5==0 \wedge notification4==0 \wedge notification==0 \wedge sendResult==0 \wedge cancel==0 \wedge meetingConfirmation==0 \wedge meetingProposition==0 \wedge medical\_examination==0)$$

Full incompatibility

When a set of n Web services do not even constitute a partial compatible choreography, we say that the services build a *fully incompatible choreography*. Full incompatibility characterizes the fact that the set of services cannot, absolutely, collaborate together. Formally, a set of services are fully incompatible, if for all the paths, all the services cannot reach the state '*final*'. This property is specified as the following CTL formulas:

$$\forall\Box\neg\, P_1.final \wedge \ldots \wedge \forall\Box\neg\, P_n.final$$

Example 13.

The PS, HAS, THS services are said to be fully incompatible if:

$$\forall\Box\neg\, PS.final \wedge \forall\Box\neg\, HAS.final \wedge \forall\Box\neg\, THS.final$$

Exprimentation

In order to validate the proposed approach, a prototype of the framework depicted in Figure 17 has been implemented in Java. CTP2UTA (Conversational Timed Protocol to UPPAAL Timed Automata) component is a parser that transforms a given timed conversational protocol into UPPAAL timed automata, according to the mapping rules described above.

Since the UPPAAL supported timed automata are described as XML document, the CTP2UTA has been implemented as a XML parser. This later was developed using the application programming interface JDOM (Java Document Object Model).

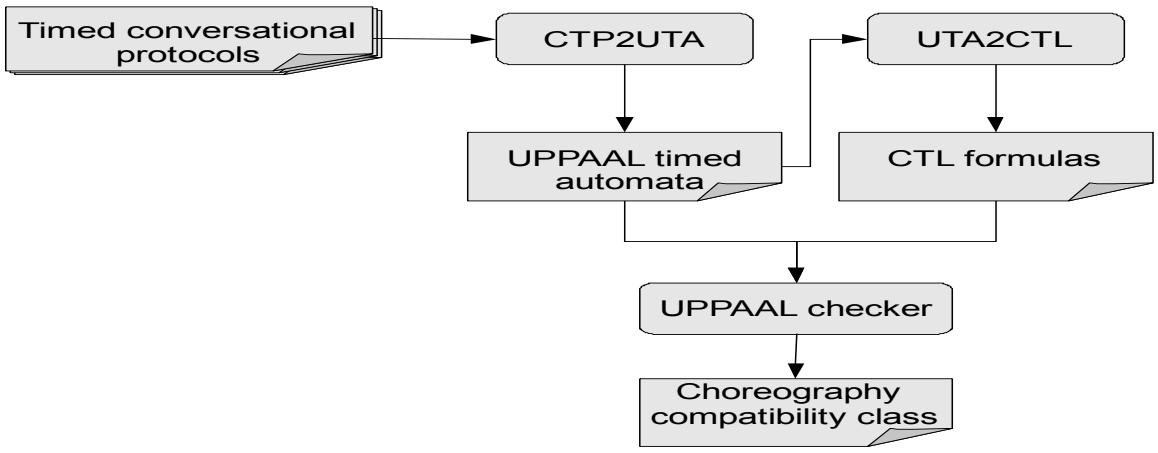

Figure 17. The underlying architecture of the compatibility framework

According to the generated UPPAAL timed automata, UTA2CTL (UPPAAL Timed Automata to CTL formulas) component generates the CTL formulas that characterize the choreography compatibility classes. These formulas associated to the generated UPPAAL timed automata of the services are then checked by the UPPAAL model checker. Figure 18 shows a snapshot of the CTP2UTA tool.

RELATED WORK

Checking and analyzing in general Web services features is an important investigation (Bordeaux and al. 2004, Benatallah-a and al. 2004-a, Benatallah-a and al. 2004-b, Benatallah and al. 2005-a, Diaz and al. 2005, Diaz and al. 2006, Ponge 2008, Kazhamiakin and al. 2006-a, Kazhamiakin and al. 2006-b, Berardi and al. 2005, Benatallah and al. 2006, Guermouche and al. 2008-a, Guermouche and al. 2008-b, Guermouche and al. 2009-a, Guermouche and al. 2009-b, Guermouche and al. 2010, Massimo and al. 2001). Particularly, in this paper we are interested in the compatibility analysis of a choreography in which services support timed asynchronous communicating services.

In (Bordeaux and al. 2004, Benatallah and al. 2004-a, Benatallah and al. 2004-b), authors consider the sequence of messages that can be exchanged between two synchronous Web services. Considering only the message exchange sequences is not sufficient. To succeed a conversation, other metrics can have an impact such as timed properties. The latter are not considered in (Bordeaux and al. 2004). Another important remark is that in (Bordeaux and al. 2004), authors consider synchronous Web services. Such assumption is very restrictive since two services can succeed a conversation in spite that they do not support the same branching structure.

The compatibility framework presented in (Ponge 2006, Ponge and al. 2007), is an extension of the framework presented in (Benatallah and al. 2005-b). It considers a more expressive timed

constraints model. Although powerful, in some cases, the compatibility framework cannot detect some timed conflicts due to non-cancellation[2] constraints.

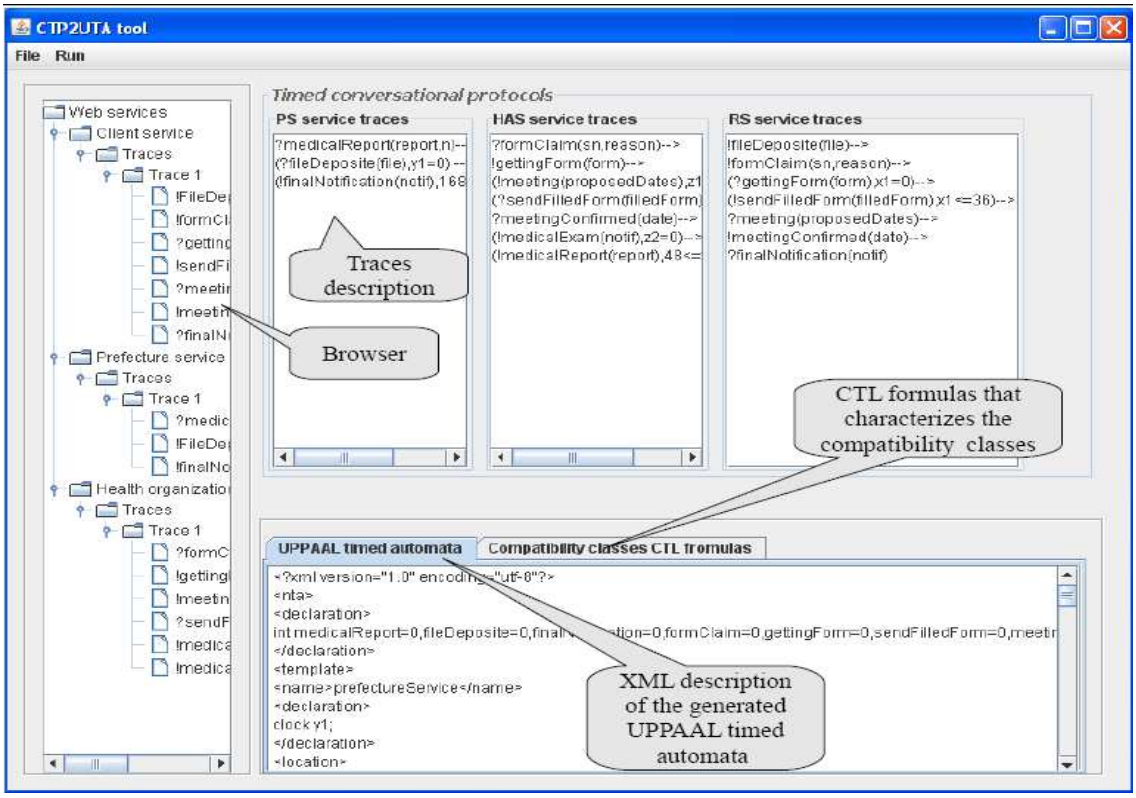

Figure 18. Screenshot of CTP2UTA tool

In fact, the authors deal only with synchronous communicating services and discovering the timed conflicts based on synchronizing the corresponding timed properties over messages. However, in case of asynchronous Web services, this framework cannot be applied to discover the eventual timed conflicts. Like the work presented in (Ponge 2006, Ponge and al. 2007), (Guermouche and al. 2008-a), handles only synchronous services.

In (Eder and al 2008), the authors handle the timed conformance problem. The timed conformance problem consists in checking if a given timed orchestration satisfies a global timed choreography. In this framework, the authors deal with timed cost (i.e., the delay) of operations. According to our work, our aim is to detect deadlocks that can arise when a set of Web services are interacting altogether. Whilst in (Eder and al. 2008), the authors are not interested in analyzing the compatibility of a choreography but only in checking if a given orchestration is conform with a choreography. So, one of the assumption on which the work presented in (Eder and al. 2008), is that the choreography does not hold timed conflicts.

Regarding our previous work, in (Guermouche and al. 2009-a, Guermouche and al. 2009-b, Guermouche and al. 2010) we present a framework for analyzing the compatibility of Web

[2] In (Ponge and al. 2007) the non-cancellation constraints are called *C-Invoke*. They specify a time window within which a given message can be fired. Outside the window, the transition is disabled (exchanging the message results in an error).

services. (Guermouche and al. 2009-b), presents an algorithm to analyze a compatibility of Web services based on the *clock ordering process* proposed in (Guermouche and al. 2008-e). This work is limited to discover only some kind of timed conflicts that do not consider other eventual timed conflicts that can arise when Web services interact together. In (Guermouche and al. 2009-a), we propose a model checking approach for analyzing the compatibility of Web services. This work considers only three general compatibility classes. Moreover, (Guermouche and al. 2009-a) do not consider data flow and data constraints when analyzing the choreography compatibility.

CONCLUSION

In this paper, we presented a formal framework for analyzing the compatibility of a choreography. Unlike the existing approaches, this framework caters for *timed properties of asynchronous* Web services. We presented how to model the timed behavior of Web services. To model timed properties, we propose to use the standard clocks of standard timed automata. In a choreography, when the services are interacting together, timed conflicts can arise. In order to handle timed deadlocks, we proposed an approach based on the model checker UPPAAL.

Since we deal with *timed asynchronous* services, the use of the *channel* notion is inadequate. This notion of channels allows to analyze only synchronous services. Moreover, UPPAAL does not take into account constraints over data semantics. In order to handle asynchronous services augmented with data flow and data constraints, we proposed a set of abstractions. By using the result of these abstractions, we presented a set of CTL formulas that characterize the different choreography compatibility classes we have defined.

In our ongoing work, we are interested in analyzing the compatibility of a choreography in which the instances of the involved services is not known in advance. Our aim is to provide primitives for defining dynamically the required instances for a successful choreography. Moreover, we plan to extend the proposed approach to support more complex timed properties when analyzing Web services.

REFERENCES

Alur, R and Dill, D. L (1994), A theory of timed automata. Theoretical Computer Science, 126(2):183–235.

Alonso, G., Casati, F., Kuno, H.A., and Machiraju, V., (2004). Web services - concepts, architectures and applications. Springer Verlag, Heidelberg.

Anca, M., and Igor, W., (2007). A lower bound on web services composition. In Proceedings of the 10th International Conference on Foundations of Software Science and Computation Structures (FOSSACS'07), volume 4423, pages 274-287, March 24-April 1, Braga, Portugal.

Benatallah, B., Casati, F., and Toumani, F., (2004-a). Web service conversation modeling : A cornerstone for e-business automation. IEEE Internet Computing, 8(1) :46-54.

Benatallah, B., Casati, F., and Toumani, F., (2004-b). Analysis and management of web service protocols. 23rd International Conference on Conceptual Modeling.

Benatallah, B., Casati, F., Ponge, J., and Toumani, F., (2005-a). On temporal abstractions of web service protocols. In The 17th Conference on Advanced Information Systems Engineering (CAiSE '05). Short Paper Proceedings.

Benatallah, B., Casati, F., Toumani, F., and Ponge, J., (2005-b). Compatibility and replaceability analysis for timed Web service protocols. In 21èmes journées Bases de données Avancées (BDA'05), October 17-20, Saint malo.

Benatallah, B., Casati, F., and Toumani, F., (2006). Representing, analysing and managing web service protocols. Data Knowedge Engineering, 58(3) :46-54.

Benatallah, B., Hamid, R., and Nezhad, M., (2007). Service oriented architecture : Overview and directions. In Lipari Summer School, pages 116-130, July 8-21, Lipari Island, Italy.

Berardi, D., (2005). Automatic Service Composition. Models, techniques and tools. PhD thesis, La Sapienza University, Roma.

Bordeaux, L., Salaûn, G., Berardi, D., and Mecella, M., (2004). When are two web services compatible? In Technologies for EServices, 5th International Workshop (TES), pages 15–28.

Bultan, T., Fu, X., Hull, R., and Su, J., (2003). Conversation specification: a new approach to design and analysis of e-service composition. In Proceedings of the international conference on World Wide Web, WWW 2003, pages 403–410.

Diaz, G., Pardo, J.-J., Cambronero, M.-E., Valero, V., and Cuartero, F., (2005). Verification of web services with timed automata. In Proceedings of the International Workshop on Automated Specification and Verification ofWeb Sites (WWV'05), volume 157 of ENTCS, pages 19–34.

Diaz, G., Cambronero, M.-E., Pardo, J.-J., Valero, V., and Cuartero, F., (2006). Automatic generation of correct web services choreographies and orchestrations with model checking techniques. In Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services (AICT/ICIW'06), page 186, 19-25 February, Guadeloupe, French Caribbean.

Dijkman., R., and Dumas, M., (2004). Service-oriented design : A multi-viewpoint approach. Technical Report CTIT Technical Report Series No. 04-09, Centre for Telematics and Information Technology, University of Twente, The Netherlands.

Eder., J., and Tahamtan, A.,. (2008). Temporal conformance of federated choreographies. In Proceedings of the 19th International Conference on Database and Expert Systems Applications (DEXA'08), Turin, Italy,September 1-5.

Guermouche, N., Perrin, O., and Ringeissen, C., (2008-a). Timed specification for web services compatibility analysis. Electr. Notes Theor. Comput. Sci., 200(3):155-170.

Guermouche, N., and Godart, C., (2008-b). Toward data flow oriented services composition. In Proceedings of the 12th International IEEE Enterprise Distributed Object Computing Conference (EDOC'08), pages 379-385, September 15-19, Munich, Germany.

Guermouche, N., Godart, C and Benatallah, B., (2008-c). Data messaging based approach for web service composition. In Proceedings of the IEEE International Conference on e-Business Engineering (ICEBE'08), pages 449-454, October 22-24, Xi'an, China.

Guermouche., N., and Godart, C., (2008-d). Uppaal based approach for compatibility analysis of synchronous web services. INRIA, Research report, 2008.

Guermouche.a., N., and Godart. C., (2008-e). Timed properties-aware asynchronous web service composition. Proceedings of the 16th International Conference on COOPERATIVE INFORMATION SYSTEMS (CoopIS'08), pages 44–61, Monterrey, Mexico, November 9-14.

Guermouche., N., and Godart., C., (2009-a). Timed model checking based approach for we bservices analysis. In Proceedings of the IEEE International Conference on Web Service (ICWS'09), July 6-10, Los Angeles, CA, USA.

Guermouche., N and Godart, C., (2009-b). Asynchronous timed web service-aware choreography analysis. In Proceedings of the 21th International Conference on Advanced Information Systems (CAiSE'09), June 8-12, Amsterdam, The Netherlands.

Henzinger, T. A., Nicollin, X., Sifakis, J., and Yovine, S., (1994). Symbolic model checking for real-time systems. Information and Computation, 111(2):193–244.

Kazhamiakin, R., Pandya, P. K., and Pistore, M., (2006-a). Representation, verification, and computation of timed properties in web service compositions. In Proceedings of the IEEE International Conference onWeb Services (ICWS), pages 497–504.

Kazhamiakin, R., Pandya, P. K., and Pistore, M., (2006-b). Timed modelling and analysis in web service compositions. In Proceedings of the The First International Conference on Availability, Reliability and Security, ARES, pages 840–846. IEEE Computer Society.

Larsen, K. G., Pettersson, P., and Yi, W., (1997). Uppaal in a nutshell. In International Journal on Software Tools for Technology Transfer.

Mecella, M., and Batini, C., (2001). Enabling italian e-government through a cooperative architecture. IEEE Computer, 34 :200-1.

Muscholl, M., and Walukiewicz, I., (2007). A lower bound on web services composition. In Proceedings of Foundations of Software Science and Computation Structures (FOSSACS), volume 4423 of LNCS, pages 274–287.

Ponge, J., (2006). A new model for web services timed business protocols. In Atelier (Conception des systèmes d'information et services Web SIWS-Inforsid).

Ponge, J, Benatallah, B., Casati, F., and Toumani, F., (2007). Fine grained compatibility and replaceability analysis of timed web service protocols. In the 26th International Conference on Conceptual Modeling (ER).

Ponge, J., (2008). Model based Analysis of Time-aware Web Services Interaction. PhD thesis, Blaise Pascal University, Clermont-Ferrand, 2008.

ABOUT THE AUTHOR

Dr. Nawal Guermouche, a postdoctoral researcher at LAAS-CNRS, Toulouse. She was a member of the SCORE team of the LORIA-INRIA research laboratory and teaching assistant at the University Henri Poincaré, of Nancy. Her research interests focus on Service Oriented Computing, particularly, timed Web services analysis and composition.

Prof. Dr. Claude Godart, is full time Professor at Nancy University, France. His center of interest concentrates on the consistency maintenance of the data mediating the cooperation between several partners. This encompasses advanced transaction models, user centric workflow and Web services composition models. He has been implicated in several transfer projects with industries (France, Europe, and Japan) for a wide range of applications including workflow, e-commerce, software processes and e-learning.